Vers la certification du compilateur v5 d'Esterel dans Coq

Delphine Kaplan-Terrasse

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Vers la certification du compilateur v5 d'Esterel dans Coq

Delphine Kaplan - Terrasse

No 4092

Décembre 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4092--FR

INRIA
SOPHIA ANTIPOLIS

Vers la certification du compilateur v5 d'Esterel dans Coq

Delphine Kaplan - Terrasse

Thème 1 — Réseaux et systèmes
Action TICK

Rapport de recherche n 4092 — Décembre 2000 — 43 pages

Résumé : Le compilateur v5 d'Esterel est basé sur un raffinement sémantique de la causalité correspondant aux "circuits constructifs" dans le domaine du matériel. La sémantique est présentée par un ensemble de règles d'inférence dans le style de la sémantique naturelle. Nous avons prouvé formellement la correction de la traduction d'une partie significative du langage en circuits au moyen du système de preuves Coq et de son interface utilisateur disponible dans Centaur.

La nouvelle sémantique constructive d'Esterel a d'abord été écrite en Typol, donnant d'une part un interprète en Prolog et d'autre part une spécification en Coq. La traduction en circuit à la base du compilateur Esterel v5 a été directement spécifiée en Coq en utilisant l'ordre supérieur. Le principal travail a consisté à certifier le compilateur v5 en Coq en raisonnant sur ces spécifications.

Mots-clé : Langages parallèles synchrones, Esterel v5, Preuves formelles, Coq, Spécification de circuits, Certification de compilateur.

(Abstract: pto)

Travail réalisé au cours d'une formation post-doctorale de janvier a juillet 1996.

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : 04 93 65 77 77 - International : +33 4 93 65 77 77
Télécopie : 04 93 65 77 65 - International : +33 4 93 65 77 65

A first step towards the proof of correctness of the v5 Esterel compiler in Coq

Abstract: The v5 compiler of Esterel is based on a refinement of causality that corresponds to the "constructive circuits" in hardware. The constructive semantics is defined by a set of inference rules like in Natural Semantics. We have formalized and proved correct the translation of a significant part of Esterel into constructive circuits, within the proof assistant Coq.

The constructive semantics of Esterel has first been written in Typol in the Centaur environment, yielding both a Prolog specification (executable) and a Coq specification. The translation of Esterel programs into circuits has been directly implemented in Coq, using higher order. The proof of correctness has then been performed, reasoning about these specifications.

Key-words: Synchronous langages, Esterel v5, Proof theory, Coq, Circuits specification, Certification of compilers.

1 Introduction

Nous décrivons dans ce rapport une première étape possible vers la certification en Coq [5] du compilateur v5 d'Esterel. Nous considérons la traduction des programmes du noyau d'Esterel (Esterel Pur) en circuits, telle qu'elle est présentée dans [2]. Cette nouvelle traduction est basée sur un raffinement sémantique de la causalité, qui correspond à la notion de *circuits constructifs* dans le domaine du matériel. Nous avons utilisé le système de preuve interactif Coq pour développer une première approximation de la preuve de correction de la traduction. Coq fournit un environnement de haut niveau, confortable pour raisonner sur des objets définis par (co)induction.

Un programme Esterel est un *module* composé d'un *ensemble de déclarations* et d'un *corps exécutable* (instruction principale). Nous nous plaçons dans le cadre d'Esterel Pur où seuls les signaux d'entrée/sortie sont déclarés et seules les instructions primitives de contrôle sont considérées (les instructions de flots de données ne sont pas prises en compte).

Nous supposerons par la suite disposer d'une infinité de signaux dont une partie finie sera utilisée par les programmes. Ceci nous permettra d'identifier un programme avec son corps en supprimant la partie déclaration.

L'implantation des programmes Esterel Pur par des circuits séquentiels, associe les entrées/sorties d'un programme avec les entrées/sorties du circuit. Chaque instant d'activation correspond à un cycle d'horloge du circuit.

Un état du circuits (donné par l'ensemble des registres) correspond pour nous à un ensemble de point d'arrêt du programme (codage binaire d'un état de l'automate de contrôle). A chaque cycle d'horloge, le circuit calcule un nouvel état et de nouvelles sorties en fonction de l'état courant et des entrées.

Pour vérifier la correction de la traduction, il faut vérifier que pour un programme donné et un événement d'entrée donné, la réaction calculée par la *sémantique constructive* est la même que celle calculée par le circuit correspondant. Nous pouvons énoncer de manière informelle cette propriété comme suit:

Un programme est *correctement implanté* par un circuit (dont les registres sont initialement à zéro), si pour tout événement d'entrée on a:

1. l'événement de sortie calculé par la sémantique constructive est identique à celui donné par le circuit,
2. le programme dérivé obtenu est *correctement implanté* par le circuit dans son nouvel état de registre.

La spécification de la propriété de correction de la traduction doit tenir compte du caractère infini du comportement d'un programme réactif et du circuit correspondant (donné par une suite infinie de réactions).

Nous nous limiterons, dans le cadre de ce rapport, à la preuve de traduction de la partie combinatoire des programmes Esterel Pur (point 1 de la définition ci-dessus).

Pour établir la correction de la traduction, nous avons choisi de nous baser sur la version *comportementale* de la sémantique constructive décrite dans le chapitre 7 de [2]. Cette sémantique *big step* nous paraît particulièrement bien adaptée aux preuves de correction. Pour formaliser la traduction nous avons utiliser la représentation graphique de l'implantation en circuits décrite dans le chapitre 11. La représentation graphique d'un circuit séquentiel est donnée par un ensemble de portes logiques et de registres reliés par des fils. Elle nous a paru la plus adéquate pour une formalisation dans Coq.

Ce rapport comporte 4 parties. La première partie traite de la formalisation de la sémantique constructive, d'une part dans le générateur d'environnement de programmation Centaur, d'autre part dans l'assistant à la preuve Coq. Nous avons ainsi obtenu dans un premier temps un environnement de programmation, qui comprend un analyseur syntaxique et un interprête pour Esterel Pur constructif. Puis nous avons formalisé cette sémantique dans Coq, en omettant dans une première approximation les instructions suivantes:

- l'instruction de boucle ("loop") pour éviter les problèmes de schizophrénie,
- les instructions d'attente ("pause") et de suspension ("suspend"), pour nous concentrer sur la partie combinatoire des circuits.

La deuxième partie discute les différentes approches possibles que nous avons envisagées pour représenter les circuits dans Coq et formaliser la traduction. Nous expliquons ici notre choix de formalisation. Dans une troisième partie nous abordons la preuve de correction proprement dite. Nous donnons en annexe B la spécification complète de la traduction dans Coq ainsi que cette preuve de correction. Enfin, dans une dernière partie, nous précisons le travail qu'il reste à faire pour complèter la preuve et nous évoquons certaines approches qui nous paraissent appropriées.

2 Spécification de la sémantique constructive

La traduction d'Esterel Pur en circuits concerne les instructions primitives de contrôle d'Esterel dont voici la liste:

nothing	0
pause	1
emit S	$!s$
present S then p else q end	$s?p,q$
suspend p when S	$s \supset p$
p ; q	$p;q$
loop p end	$p*$
p || q	$p\|q$
trap T in p end	$\{p\}$
	$\uparrow p$
exit T	k avec $k \geq 2$
signal S in p end	$p\backslash s$

Nous avons indiqué à droite la syntaxe héritée du calcul de processus qui est utilisée dans [2] pour décrire la sémantique constructive. Ces instructions sont impératives et leur comportement intuitif est expliqué dans le ch.2 de [2]. Nous rappelons simplement les principales différences avec la syntaxe d'origine du noyau:

- l'instruction "halt", qui ne termine jamais, est remplaçée par l'instruction "pause" qui se termine à l'instant suivant (halt ≡ loop pause end),

- l'instruction "do p watching S" utilisée pour interrompre une instruction p à partir des instants qui suivent le premier instant, sans l'exécuter (préemption forte), est remplaçée par l'instruction "suspend p when S", qui suspend l'instruction p,

- le signal tick, toujours présent, peut être introduit par un signal local, en exécutant l'instruction suivante en parallèle:

```
loop
   emit tick;
   pause
end.
```

L'opérateur "↑" (shift) est utilisé dans la définition des instructions dérivée d'Esterel, pour décaler le code de terminaison d'une sous-instruction englobée par un "trap". Par exemple, l'instruction "do p watching S" est codée par: $s \gg p \equiv \{(s \supset\uparrow p; 2)|(1; s?2, 0)*\}$. Nous utilisons la syntaxe "calcul de processus" dans ce qui suit par soucis de concision. Par exemple le programme:

```
trap T in
      pause
   ||
      exit T
end
```

sera noté tout simplement $\{1|2\}$. Les codes de terminaison $k \geq 2$ correspondent à l'indice de DeBruijn (augmenté de 2) associé à l'introduction des exceptions. Dans l'exemple précédant, l'indice est 2 $(0 + 2)$, puisqu'il se réfère à la première (et unique dans ce cas) exception rencontrée en remontant dans le programme.

La sémantique constructive d'Esterel est donnée dans [2] sous les trois formes habituelles: la sémantique constructive comportementale, la sémantique constructive opérationnelle et la sémantique constructive par états - ou "ensemble de points d'arrêt". Nous détaillons dans ce qui suit la spécification du langage Esterel dans Centaur, puis sa spécification dans le système Coq. Ces deux spécifications nous conduirons d'une part à un environnement de programmation opérationnel, d'autre part à un environnement de preuve.

2.1 Spécification dans Centaur

Nous avons décrit le noyau d'Esterel dans Centaur en utilisant la sémantique constructive comportementale par état. Cette sémantique utilise une définition étendue des termes (termes "décorés") qui prend en compte la notion de points d'arrêts, permettant ainsi le codage des états sur le programme initial. Autrement dit, c'est une sémantique par ensemble de points d'arrêt qui permet de visualiser la progression des états sur le programme initial. Dans sa version comportementale, cette sémantique se prête particulièrement bien au suivi de sujet disponible en sémantique naturelle. Ceci permet entre autre de visualiser les instructions activées dans l'instant et de répondre à des besoins de debug.

L'environnement développé dans Centaur propose de charger un programme décoré et de lui appliquer une transition pour un événement d'entrée donné. Le nouveau programme décoré apparaît alors à la place de l'ancien. L' environnement que je viens d'évoquer est opérationnel. Il gagnerait toutefois à être complété de la façon suivante:

- description de la sémantique constructive comportementale en sémantique naturelle,
- traduction automatique des deux spécifications comportementales (avec et sans état) dans Coq,
- preuve de l'équivalence de ces deux spécifications dans Coq.

Il reste également des efforts à faire au niveau du parser (seul la syntaxe "calcul de processus" est décrite), ainsi qu'au niveau de l'extension de l'environnement, pour prendre en compte les instructions flots de données et les instructions dérivées.

Comme nous allons le voir maintenant, nous avons utilisé la sémantique constructive comportementale pour spécifier la traduction des programmes Esterel en circuits dans Coq.

2.2 Spécification dans Coq

Les instructions d'Esterel Pur sont décrites dans Coq par l'ensemble inductif `Stmt`. Nous avons codé les signaux par les entiers, d'où l'argument entier qui apparaît dans les instructions d'émission et de présence.

```
Inductive nat : Set := O: nat | S: nat->nat.

Inductive Stmt: Set :=
   completion: nat ->Stmt
  | emit: nat ->Stmt
  | present: nat -> Stmt -> Stmt ->Stmt
  | seq: Stmt -> Stmt ->Stmt
  | parallel: Stmt -> Stmt ->Stmt
  | local: nat -> Stmt ->Stmt
  | trap: Stmt ->Stmt
  | shift: Stmt ->Stmt.
```

Nous avons omis volontairement l'instruction de boucle (loop) qui est à l'origine des problèmes de schizophrénie décrits dans le ch.12 de [2]. Nous ne traitons donc pas la traduction des boucles en première approximation. Nous avons également omis l'instruction de suspension (suspend) qui ne présente pas d'intérêt au premier instant. Nous nous plaçons en effet dans le premier instant, où seule la partie combinatoire des circuits séquentiels intervient.

Nous allons donner maintenant la formalisation de la sémantique constructive comportementale dans Coq.

Comme nous l'avons déjà vu, les signaux sont codés par les entiers. Pour simplifier, nous avons décider de coder les statuts associés aux signaux par les booléens de Scott. On notera $\mathcal{B}_\perp = \{\perp, \mathbf{0}, \mathbf{1}\}$ le domaine de Scott. Dans ce contexte, l'élément $\mathbf{0}$ dénote l'absence d'un signal, l'élément $\mathbf{1}$ dénote la présence d'un signal et l'élément $\perp$ dénote une valeur indéterminée pour un signal. Le domaine $\mathcal{B}_\perp$ est spécifié dans Coq par un ensemble inductif:

```
Inductive ScottBool: Set :=
   zero: ScottBool
  | one: ScottBool
  | bottom: ScottBool.
```

Nous définissons ensuite un événement comme une fonction des entiers (signaux) dans les booléens de Scott (statuts). Les booléens de Scott nous permettront de représenter les événements partiels, qui apparaissent lors du traitement des signaux locaux.

Nous avons défini les *Scott-adiques* par analogie avec les *2-adiques*, comme étant des suites infinies de booléens de Scott. Les *Scott-adiques* peuvent être spécifiés dans Coq par les fonctions des entiers dans les booléens de Scott:

```
Definition ScottAdic: Set := nat ->ScottBool.
```

Ce type de donnée nous sera également trés utile pour décrire les circuits et nous le détaillons les dans ce qui suit.

Nous notons Zero le Scott-adique particulier qui est nul partout "Zero $\equiv \lambda n.\mathbf{0}$". Les opérateurs Or, And et Not sur les Scott-adiques sont les extensions naturelles des opérateurs *or*, *and* et *not* définis sur le domaine de Scott $\mathcal{B}_\perp$ (fig.1).

or	0	1	$\perp$
0	0	1	$\perp$
1	1	1	1
$\perp$	$\perp$	1	$\perp$

and	0	1	$\perp$
0	0	0	0
1	0	1	$\perp$
$\perp$	0	$\perp$	$\perp$

	0	1	$\perp$
not	1	0	$\perp$

FIG. 1 – *Opérateurs sur les booléens de Scott*

Soient A, B deux Scott-adiques et n un naturel, les différents opérateurs sont étendus comme suit:

$$\begin{array}{lcl} A \ \mathsf{Or} \ B & \equiv & \lambda n.\ A(n)\ or\ B(n) \\ A \ \mathsf{And} \ B & \equiv & \lambda n.\ A(n)\ and\ B(n) \\ \mathsf{Not}\ A & \equiv & \lambda n.\ not\ A(n) \end{array}$$

Nous donnons également la définition des opérateurs de décalages et de maximum, étendus aux Scott-adiques:

$$\downarrow A \equiv \ \lambda n. \begin{cases} A(0)\ or\ A(2) & \text{si } n = 0 \\ A(1) & \text{si } n = 1 \\ A(n+1) & \text{si } n \geq 2 \end{cases}$$

$$\uparrow A \equiv \lambda n. \begin{cases} A(n) & \text{si } n = 0 \text{ ou } n = 1 \\ \mathbf{0} & \text{si } n = 2 \\ A(n-1) & \text{si } n > 2 \end{cases}$$

Nous rappelons que le maximum de deux ensembles de naturels consiste en leur union, à laquelle on retranche l'ensemble des naturels qui sont strictement inférieurs au maximum des deux minimums.

Pour calculer le maximum de deux Scott-adiques A et B, il suffit de dépasser le premier naturel qui a pour valeur **1** dans A, ainsi que le premier naturel qui a pour valeur **1** dans B, puis d'avoir la valeur **1** soit dans A soit dans B. Ceci nous donne la formule possible suivante:

$$\mathsf{Max}(A,B) \equiv \lambda n. \begin{array}{c} A(0)\ or\ A(1)\ or\ \ldots\ or\ A(n) \\ and \\ A(n)\ or\ B(n) \\ and \\ B(0)\ or\ B(1)\ or\ \ldots\ or\ B(n) \end{array}$$

Tout ces opérateurs sur les Scott-adiques ont une spécification très naturelle dans Coq que nous donnons en annexe B.

Une autre opération importante est la mise à jour des Scott-adiques lorsque l'on veut changer la valeur affectée à un entier donné. Celle-ci se fera grace à la fonction "New" qui prend en argument un Scott-adique E, un naturel n ainsi qu'un booléen de Scott b et retourne le nouveau Scott-adique (noté $E[n \mapsto b]$) pour lequel n est associé à b.

$$E[n \mapsto b] \equiv \lambda m. \begin{cases} E(m) & \text{si } m \neq n \\ b & \text{si } m = n \end{cases}$$

Nous avons besoin, pour définir cette fonction de mise à jour, d'un petit lemme arithmétique "Lemnat" qui permet de décider de l'égalité de deux naturels.

```
Lemma Lemnat: (n,m:nat){n=m}+{~ n=m}.

Definition New: ScottAdic -> nat -> ScottBool -> ScottAdic :=
   [f:ScottAdic] [i:nat] [x:ScottBool] [n:nat]
   (<ScottBool>Case (Lemnat n i) of [H:n=i]x [H:~ n=i](f n) end).
```

Les événements vont pouvoir être représentés par des éléments du type `ScottAdic`.

Nous donnons à présent la spécification de la sémantique constructive comportementale. Nous avons légèrement modifié la définition des prédicats *Must* et *Can* (ch.7 de [2]) pour les englober dans la définition d'un unique prédicat *MC*. Ce dernier est paramétré par un booléen de Scott (α) qui joue un rôle de contrôle, comme c'est le cas pour le prédicat *Can* et son *must code*.

Le prédicat *MC* a la forme suivante:

$$MC^{\alpha}(p,E) \;=\; < S, K > \;=\; < MC^{\alpha}_{s}(p,E), MC^{\alpha}_{k}(p,E) >$$

où S est un Scott-adique qui dénote un événement partiel et K un Scott-adique qui dénote un ensemble de codes de complétion. Le contrôle α est à $\mathbf{0}$ si on *ne peut pas* passer par l'instruction p, il est à $\mathbf{1}$ si on *doit* y passer et à $\perp$ si on *peut* y passer mais qu'on *ne doit pas* le faire impérativement. Si le contrôle est à $\mathbf{1}$, on récupère les signaux/code qui *doivent* être émis/retourné (statut $\mathbf{1}$) et les signaux/codes qui *ne peuvent pas* être émis/retournés (statut $\mathbf{0}$), les signaux/codes qui appartiennent aux complémentaires ayant le statut $\perp$. Si le contrôle est à $\perp$, on récupère seulement les signaux/codes qui *ne peuvent pas* être émis/retournés (valeur $\mathbf{0}$), les signaux/codes qui appartiennent aux complémentaires ayant le statut $\perp$.

Le prédicat *MC* peut être obtenu à partir d'un codage de *Must* et *Can* que nous définissons informellement par:

- $MC^{\mathbf{0}}(p,E) =< \mathsf{Zero}, \mathsf{Zero} >$
- si $\alpha \neq \mathbf{0}$ $MC^{\alpha}(p,E) = \lambda n.$ $\begin{cases} \alpha \text{ si } n \in Must(p,E) \\ \mathbf{0} \text{ si } n \in \overline{Can^{\alpha}}(p,E) \\ \perp \text{ si } n \in \overline{Must}(p,E) \cap Can^{\alpha}(p,E) \end{cases}$

On peut également visualiser ce codage par le shéma de la figure 2.

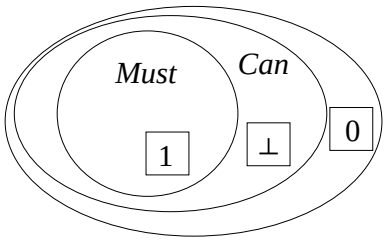


FIG. 2 – *Codage des prédicats Must et Can par les booléens de Scott*

Nous pouvons donner maintenant la définition formelle de *MC* pour $\alpha \neq \mathbf{0}$, qui est défini inductivement sur les instructions. Comme nous l'avons déjà souligné, nous partons de la définition de *Must* et *Can* donnée dans le ch.7 de [2].

Les opérateurs sur les Scott-adiques seront implicitement surchargés pour pouvoir s'appliquer à une paire de Scott-adiques.

Dans le cas d'un code de complétion k, aucun signal *ne peut être* émis et les codes de complétion $k' \neq k$ *ne peuvent pas* non plus être atteint ($k' \in \overline{Can^\alpha}(k, E)$ pour $k' \neq k$). Le code k *doit* être atteint si nous devons passer par cette instruction (contrôle $\alpha = \mathbf{1}$). Il *peut* être atteint mais *ne doit pas* l'être si nous ne sommes pas obligés d'y passer (contrôle $\alpha = \bot$). Nous avons donc par définition:

$$MC^\alpha(k, E) =< \mathsf{Zero}, \mathsf{Zero}[k \mapsto \alpha] >$$

En raisonnant de façon similaire pour les émissions de signaux nous obtenons:

$$MC^\alpha(!s, E) =< \mathsf{Zero}[s \mapsto \alpha], \mathsf{Zero}[0 \mapsto \alpha] >$$

Le seul code de terminaison qui peut être atteint est 0 car une émission termine toujours.

Pour le test de présence, les cas où le signal est présent ou absent sont simples: il suffit de rappeller le prédicat récursivement. Lorque le signal est indéterminé, aucun signal ne doit être émis et aucun code de terminaison ne doit être atteint. Il nous faut donc imposer un contrôle $\alpha = \bot$ sur tout appel récursif. Les signaux/codes qui ne peuvent pas être émis/retournés sont dans l'intersection de ceux qui ne peuvent pas l'être pour les sous-instructions. Ceci se traduit facilement grace à l'opérateur Or sur les Scott-adiques (en effet $a \; or \; b = \mathbf{0} \Leftrightarrow a = \mathbf{0} \text{et} b = \mathbf{0}$). Nous avons donc:

$$MC^\alpha(s?p, q\ , E) = \begin{cases} MC^\alpha(p, E) \text{ si } E(s) = \mathbf{1} \\ MC^\alpha(q, E) \text{ si } E(s) = \mathbf{0} \\ MC^\bot(p, E) \ \mathsf{Or} \ MC^\bot(q, E) \text{ si } E(s) = \bot \end{cases}$$

Pour la séquence, trois cas doivent également être envisagés selon la valeur du code de terminaison de la première instruction passé en contrôle à la deuxième instruction. Il est important, dans le cas où cette valeur est $\bot$, de rappeler récursivement MC avec $\bot$ sur la deuxième instruction.

$$MC^\alpha(p; q\ , E) = \begin{cases} < MC^\alpha_s(p, E) \ \mathsf{Or} \ MC^\alpha_s(q, E)\ , & \\ \quad MC^\alpha_k(p, E)[0 \mapsto \mathbf{0}] \ \mathsf{Or} \ MC^\alpha_k(q, E) > & \text{si } MC^\alpha_k(p, E)(0) = \mathbf{1} \\ MC^\alpha(p, E) & \text{si } MC^\alpha_k(p, E)(0) = \mathbf{0} \\ < MC^\alpha_s(p, E) \ \mathsf{Or} \ MC^\bot_s(q, E)\ , & \\ \quad MC^\alpha_k(p, E)[0 \mapsto \mathbf{0}] \ \mathsf{Or} \ MC^\bot_k(q, E) > & \text{si } MC^\alpha_k(p, E)(0) = \bot \end{cases}$$

Pour l'instruction parallèle, on utilise tout simplement l'appel récursif du prédicat sur les deux sous-instructions, que l'on combine avec les opérateurs Or et Max:

$$MC^\alpha(p|q\ , E) =< MC^\alpha_s(p, E) \ \mathsf{Or} \ MC^\alpha_s(q, E),\ \mathsf{Max}(MC^\alpha_k(p, E), MC^\alpha_k(q, E)) >$$

La définition du prédicat pour les instructions d'exception et de shift est triviale:

$$MC^{\alpha}(\{p\}\ , E) =< MC^{\alpha}_{s}(p, E)\ ,\ \downarrow MC^{\alpha}_{k}(p, E) >$$

$$MC^{\alpha}(\uparrow p\ , E) =< MC^{\alpha}_{s}(p, E)\ ,\ \uparrow MC^{\alpha}_{k}(p, E) >$$

Pour la déclaration d'un signal local, trois cas doivent être étudier selon un premier calcul de MC avec un signal s indéterminé. Le premier cas intervient lorsque l'on *doit* passer par l'instruction et que le signal s *doit* être émis: on appelle alors récursivement MC avec le statut de présence associé à s. Le deuxième cas apparait lorsque le signal déclaré *ne peut pas* être émis dans l'instruction, quelque soit la valeur du contrôle: on appelle alors récursivement MC avec le statut d'absence associé à s. Enfin, si le statut de s reste indéterminé aprés le calcul de MC, on garde ce dernier appel récursif. On peut résumer ces trois cas ainsi:

- on propage les "must" ($\mathbf{1}$) lorque l'on doit passer par l'instruction (contrôle à $\mathbf{1}$),

- on propage les "$\overline{\text{can}}$" ($\mathbf{0}$) lorque l'on peut passer par l'instruction (contrôle à $\mathbf{1}$ ou $\perp$),

- on ne propage rien sinon.

$$MC^{\alpha}(p\backslash s\ , E) = \begin{cases} MC^{\mathbf{1}}(p, E[s \mapsto \mathbf{1}]) & \text{si } \alpha = \mathbf{1} \text{ et } MC^{\mathbf{1}}_{s}(p, E[s \mapsto \perp])(s) = \mathbf{1} \\ MC^{\alpha}(p, E[s \mapsto \mathbf{0}]) & \text{si } MC^{\alpha}_{s}(p, E[s \mapsto \perp])(s) = \mathbf{0} \\ MC^{\alpha}(p, E[s \mapsto \perp]) & \text{si } MC^{\alpha}_{s}(p, E[s \mapsto \perp])(s) = \perp \end{cases}$$

Notons que le cas où $\alpha = \perp$ et s associé à $\mathbf{1}$ est impossible. On pourra vérifier plus généralement que le prédicat MC ne peut engendrer de $\mathbf{1}$ si $\alpha = \perp$.

Remarque Le codage de *Must* et *Can* qui permet d'obtenir MC mériterait d'être formalisée plus précisément. La preuve de $MC^{\perp}(p, E)(x) \neq \mathbf{1}$ pourrait être faite dans Coq.

La spécification du prédicat MC dans Coq est obtenue directement à partir de la définition vue précédemment. Il s'agit d'un prédicat défini inductivement dont voici le début:

```
Inductive MC: ScottBool -> Stmt -> ScottAdic -> ScottAdic -> ScottAdic ->Prop :=
   MC_alpha0: (E:ScottAdic) (p:Stmt)(MC zero p E Zero Zero)
  | MC_compl:
      (alpha:ScottBool)(k:nat)(E:ScottAdic) ~ alpha =zero ->
      (MC alpha (completion k) E Zero (New Zero k alpha))
  | MC_present1:
      (alpha:ScottBool)(si:nat)(p,q:Stmt)(E,E',K:ScottAdic)
      ~ alpha =zero -> (E si) =one -> (MC alpha p E E' K) ->
      (MC alpha (present si p q) E E' K)
  :
  :
  :
```

La relation de transition $p \overset{E',k}{\underset{E}{\hookrightarrow}} p'$ donnée dans le ch 7 de [2] est également spécifiée dans Coq par un prédicat défini inductivement et ne pose pas de problème particulier. Nous en donnons les premières lignes:

```
Inductive TRANSITION: ScottAdic -> Stmt -> Stmt -> ScottAdic -> nat ->Prop :=
   trans_compl:
     (E:ScottAdic)(k:nat)(TRANSITION E (completion k) (completion 0) Zero k)
  | trans_emit:
      (si:nat)(E:ScottAdic)
      (TRANSITION E (emit si) (completion 0) (New Zero si one) 0)
  | trad_present:
      (si:nat) (p, q:Stmt) (P, Q:Circ) (TRAD p P) -> (TRAD q Q) ->
      (TRAD
         (present si p q)
         [GO:ScottBool] [E:ScottAdic]
         <ScottAdic, ScottAdic>
             ((Or
                 (Fst (P (Sand GO (E si)) E))
                 (Fst (Q (Sand GO (Snot (E si))) E))),
              (Or
                 (Snd (P (Sand GO (E si)) E))
                 (Snd (Q (Sand GO (Snot (E si))) E)))))

  :
  :
  :
```

Toute la spécification dans Coq de la sémantique constructive comportementale d'Esterel Pur (sans boucle et sans suspension) est donnée en annexe B. La relation de transition ne nous servira pas pour la preuve de correction, que nous avons restreinte à la partie combinatoire de la traduction en circuit. Elle nous sera utile pour évoquer l'extension de la preuve à la partie registre (prise en compte de l'attente et de la suspension) dans le ch.5.

3 Spécification de la traduction dans Coq

Nous allons tout d'abord briévement rappeler les différentes solutions que nous avons étudiées, avant de choisir celle qui nous a paru la meilleure pour une implantation dans Coq.

Une première approche qui vient à l'esprit part de la traduction formalisée dans le ch.13 de [2]. Cette traduction est définie récursivement sur les programmes et associe à chaque programme constructif un ensemble d'équation booléennes. Sa formalisation est trés proche de l'interprête implanté en C. Elle nécessite une gestion explicite des noms de fils, incluant les problèmes de renommage, substitution, etc.

Nous avons préféré nous diriger vers une approche plus abstraite, où les fils sont représentés par des abstractions. En utilisant l'ordre supérieur disponible dans Coq, nous pourrons

ainsi bénéficier des opérations de renommage et substitution, gérées une fois pour toute au niveau méta.

Deux solutions ont été envisagées selon cette deuxième approche, qui s'inspirent toute deux de la représentation graphique des circuits donnée dans le ch.11.

La première est algébrique: elle décrit un circuit comme une structure récursive dépendant de deux entiers. Le premier représente le nombre n de fils entrant et le deuxième le nombre m de fils sortant. La sémantique associée à un circuit est alors définie récursivement sur les circuits, et retourne une fonction de n valeurs booléennes vers m valeurs booléennes.

La deuxième solution est dénotationnelle: elle décrit un circuit comme une fonction. Nous supposerons une infinité de fils entrants et sortants. Cette dernière solution a servi de base au développement de la preuve de traduction dans Coq, que nous présenterons dans le prochain chapitre.

Un circuit qui implante un programme Esterel a le shéma suivant:

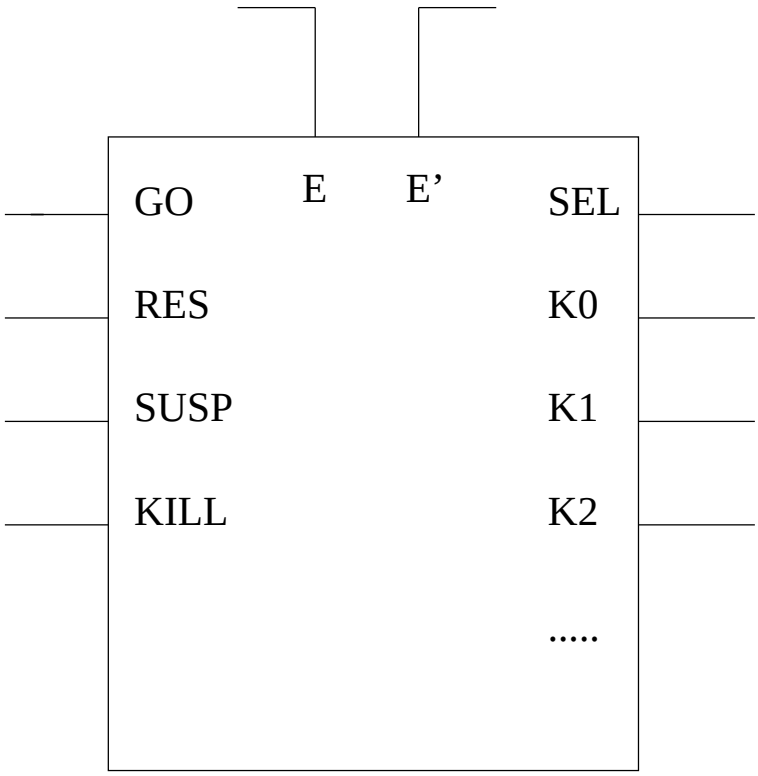


FIG. 3 – *Shéma d'un circuit séquentiel implantant un programme Esterel Pur*

Dans le cadre de ce travail nous ne traitons pas les instructions d'attente et de suspension qui se traduisent par des circuits avec registres. Ceci afin de pouvoir nous concentrer sur la partie combinatoire de la traduction en circuits.

Nous pouvons donc omettre ici les fils RES, SUSP, KILL et SEL. Nous rappelons toutefois leurs rôles respectifs:

- le fil RES (resume) contrôle l'activation au cours des instants suivant le premier instant,

- le fil SUSP gèle l'état des registres, notons que RES est alors déactivé (RES=$\neg$SUSP),

- le fil KILL remet les registres à zéro lors d'une levée d'exception,

- le fil SEL est la disjonction des registres.

Les fils que nous retenons en première approximation sont les suivants:

- le fil GO qui contrôle l'activation initiale (il sera implanté plus tard par un registre),

- les fils correspondant aux signaux d'entrée (E),

- les fils correspondant aux sigaux de sortie (E') et les fils correspondant aux codes de terminaison (K0,K1,K2,etc).

D'où le shéma simplifié donné en figure 4 que nous avons utilisé.

Nous verrons par la suite qu'il sera facile d'étendre les circuits avec les fils manquants. Nous donnons dans la fig.5 les circuits qui implantent les instructions Esterel permises dans le cadre de notre étude. Nous notons K_i lorsque nous voulons isoler un fil, auquel cas K représente les K_j pour j $\neq$ i.

3.1 1^{ere} approche

Nous donnons uniquement un aperçu de cette approche qui a une formalisation plutôt lourde dans Coq.

Un circuit est représenté ici par une structure récursive de portes. Une porte dépend de deux naturel n (nombre d'entrées) et m (nombre de sorties). Une porte peut être un buffer, une porte OU, une porte ET, une porte NOT, une composition de portes, un produit de portes, etc. Soit $a: \ n \rightarrow m$ (resp. $b: \ p \rightarrow q$) une porte à n (resp. p) fils entrants et n (resp. q) fils sortants. Une porte est construite à partir des opérateurs élémentaires suivants:

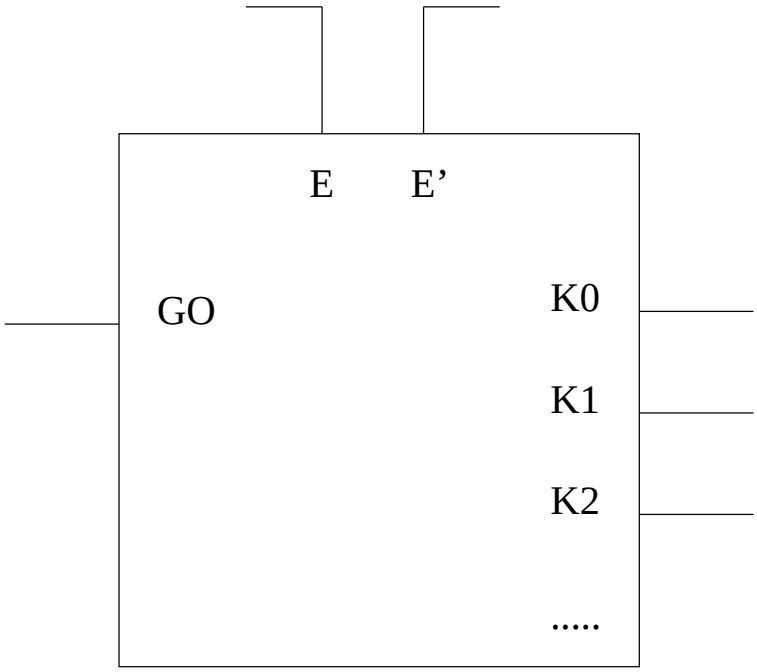
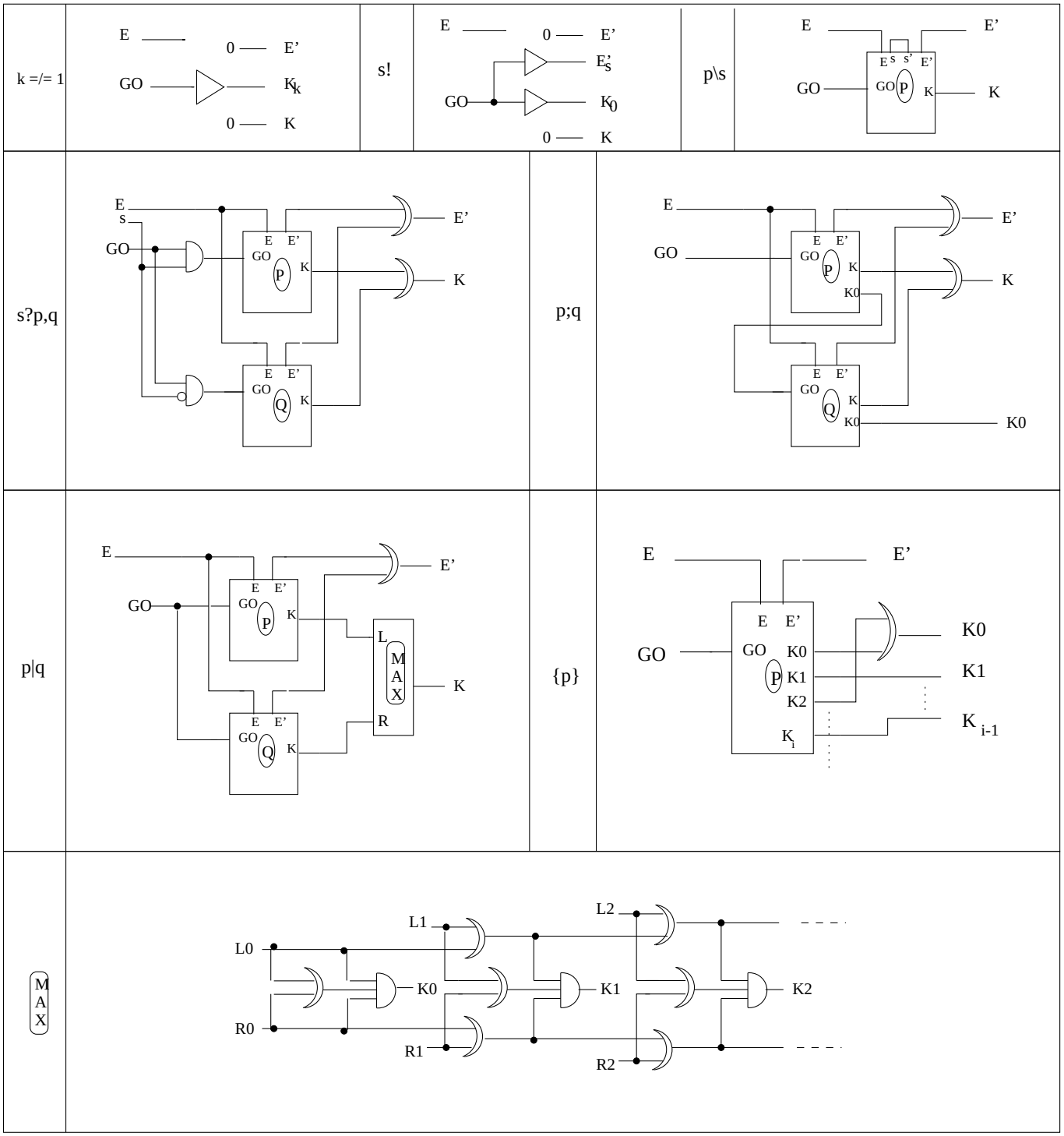


FIG. 4 – *Shéma des circuits sans registres*



RR n 4092 FIG. 5 – *Circuits implantant les instructions d'Esterel traitées*

$\sharp 0$	off
$\sharp 1$	on
$\vee$	porte OU
$\wedge$	porte ET
$\neg$	porte NOT
I	identité
$\triangle$	duplication
$a; b$	compositon
$a \mid b$	produit
$a*$	réflexion
μa	permutation

La sémantique d'un circuit sera alors donnée par une fonction dans $\mathcal{B}_\perp^n \rightarrow \mathcal{B}_\perp^m$. Nous aurons besoin dans un premier temps de spécifier le produit cartésien de booléen de Scott. Nous donnons la spécification dans Coq des circuits et de la traduction suivant cette approche en annexe A. Cette formalisation n'est pas complète notament en ce qui concerne le codage de la réflexion (calcul de point fixe).

3.2 2^{eme} approche

Dans cette approche, les fils sont interprétés dans le cpo des booléens de Scott $\mathcal{B}_\perp$. Les fils correspondant aux signaux d'entrée (E), aux signaux de sortie (E') et aux codes de terminaison (K), sont représentés par des Scott-adiques. Ceci nous permet d'avoir une écriture uniforme pour dénoter à la fois un ensemble de fils, un événement et un ensemble de codes de terminaison.

Un circuit est représenté par une fonction qui prend en arguments un booléen de Scott (GO), un Scott-adique (E), et retourne une paire de Scott-adiques (E',K).

La spécification des circuits dans Coq sera donc:

```
Definition Circ: Set := ScottBool -> ScottAdic -> ScottAdic * ScottAdic.
```

Nous allons donner à présent les règles de la traduction qui découlent de l'implantation des programmes en circuits donnée dans la fig.5. Nous donnons ces règles dans le style de la sémantique naturelle, par soucis de clareté.

Les symboles p et q dénotent des programmes Esterel, P et Q des circuits, et go et E dénotent respectivement un booléen de Scott et un Scott-adique.

Si X est la paire $< A, B >$, on note X_1 son premier élément (A) et X_2 son deuxième élément (B).

Enfin on note $\vee, \wedge$ et $\neg$ les opérateurs *or*, *and* et *not* sur les booléens de Scott.

trad_compl : $k \rightarrow \lambda go.\lambda E. < \mathsf{Zero} , \mathsf{Zero}[0 \mapsto go] >$

trad_emit : $s! \rightarrow \lambda go.\lambda E. < \mathsf{Zero}[s \mapsto go] , \mathsf{Zero}[0 \mapsto go] >$

trad_present :
$$\frac{p \rightarrow P \qquad q \rightarrow Q}{s?p,q \rightarrow \lambda go.\lambda E. < (P\ go \wedge E(s)\ E)_1\ \mathsf{Or}\ (Q\ go \wedge \neg E(s)\ E)_1 , (P\ (go \wedge E(s))\ E)_2\ \mathsf{Or}\ (Q\ (go \wedge \neg E(s))\ E)_2 >}$$

trad_seq :
$$\frac{p \rightarrow P \qquad q \rightarrow Q}{p;q \rightarrow \lambda go.\lambda E. < (P\ go\ E)_1\ \mathsf{Or}\ (Q\ ((P\ go\ E)_2\ 0)\ E)_1 , \lambda n. \left\{ \begin{array}{l} (Q\ ((P\ go\ E)_2\ 0)\ E)_2(0) \text{ si } n = 0 \\ ((P\ go\ E)_2\ \mathsf{Or}\ (Q\ ((P\ go\ E)_2\ 0)\ E)_2)(n) \text{ si } n \neq 0 \end{array} \right\rangle}$$

trad_parallel :
$$\frac{p \rightarrow P \qquad q \rightarrow Q}{p|q \rightarrow \lambda go.\lambda E. < (P\ go\ E)_1\ \mathsf{Or}\ (Q\ go\ E)_1 , \mathsf{Max}((P\ go\ E)_2, (Q\ go\ E)_2) >}$$

trad_local :
$$\frac{p \rightarrow P}{p\backslash s \rightarrow \lambda go.\lambda E.(P\ go\ E[s \mapsto (P\ go\ E[s \mapsto \bot])_1(s)])}$$

trad_trap :
$$\frac{p \rightarrow P}{\{p\} \rightarrow \lambda go.\lambda E. < (P\ go\ E)_1 , \downarrow (Q\ go\ E)_2 >}$$

trad_shift :
$$\frac{p \rightarrow P}{\uparrow p \rightarrow \lambda go.\lambda E. < (P\ go\ E)_1 , \uparrow (Q\ go\ E)_2 >}$$

FIG. 6 – *Règles de la traduction en circuit*

Ces règles ont été trivialement obtenues à partir de la représentation graphique des circuits implantant les instructions Esterel (fig.5). La règle du signal local mérite toutefois une explication. La traduction du signal local devrait en fait avoir la forme suivante:

$$\lambda go.\lambda E.(P\ go\ E[s \mapsto fix(\lambda X.\ (P\ go\ E[s \mapsto X])_1(s))$$

où s a pour valeur le point fixe de la fonction qui prend en argument le statut de s et retourne le nouveau statut associée à s par le circuit. Cela revient en quelque sorte à *currifier* le calcul du point fixe de l'ensemble des signaux locaux dans le circuit.

Pour obtenir la règle de traduction de la déclaration locale, nous avons pris comme point fixe de la fonction

$$\lambda X.\ (C\ go\ E[s \mapsto X])_1(s)$$

sa valeur pour $X = \perp$. Nous expliquons, dans les quelques lignes qui suivent, les raisons pour lesquelles le point fixe d'une fonction continue sur $\mathcal{B}_\perp$ peut être obtenu en une itération sur $\perp$.

Rappelons que les fonctions continues sur le cpo $\mathcal{B}_\perp$, muni de l'ordre partiel $\perp \leq \mathbf{0}$ et $\perp \leq \mathbf{1}$, sont les fonctions monotones sur $\mathcal{B}_\perp$ ($continue \leftrightarrow monotone\ +\ finitaire$). Cette continuité peut s'exprimer de manière très simple dans le cas particulier du cpo de Scott par la propriété suivante:

$$f : \mathcal{B}_\perp \rightarrow \mathcal{B}_\perp \text{ est continue } \equiv\ f(\perp) \neq \perp\ \rightarrow\ f(\mathbf{0}) = f(\mathbf{1}) = f(\perp)$$

En effet, f est continue $\leftrightarrow f$ est monotone
$\leftrightarrow f(\perp) \leq f(\mathbf{0})$ et $f(\perp) \leq f(\mathbf{1})$
$\leftrightarrow (f(\perp) = \perp)$ ou $(f(\perp) \neq \perp$ et $f(\mathbf{0}) = f(\mathbf{1}) = f(\perp))$. .

Si f est continue, d'aprés le théorème de Tarski il existe un plus petit point fixe qui s'obtient par itération à partir de $f(\perp)$. On peut s'arrêter à la première itération dans ce cas. Nous avons supposer ici que la fonction "$\lambda X.\ (C\ go\ E[s \mapsto X])_1(s)$" est continue (les circuits impliqués dans la traduction étant composés à partir des connecteurs *or*, *and* et *not* qui sont eux-même continus).

Les règles de traduction données plus haut définissent un prédicat qui est spécifié inductivement dans Coq de manière naturelle. Nous en donnons le début, le reste pouvant être consulté en annexe B.

```
Inductive TRAD: Stmt -> Circ ->Prop :=
   trad_compl:
     (k:nat)
     (TRAD
        (completion k)
        [GO:ScottBool] [E:ScottAdic]
        <ScottAdic, ScottAdic> (Zero, (New Zero k GO)))
  | trad_emit:
```

```
      (si:nat)
      (TRAD
         (emit si)
         [GO:ScottBool] [E:ScottAdic]
         <ScottAdic, ScottAdic> ((New Zero si GO), (New Zero O GO)))
  :
  :
  :
```

Une autre solution pourrait prendre en compte de manière explicite la continuité de "$\lambda X.\ (C\ go\ E[s \mapsto X])_1(s)$". Pour pouvoir utiliser le point fixe de cette fonction, au niveau de la traduction du signal local, on pourrait ajouter la preuve de cette continuité *dans* la définition de la traduction. Il s'agit de rajouter un argument de continuité de la fonction

$$\lambda X.\ (C\ Y\ E[s \mapsto X])_1(s)$$

au prédicat qui spécifie la traduction. Mais il faudra également rajouter les arguments pour la continuité des fonctions suivante (à cause de la séquence):

$$\lambda Y.\ (C\ Y\ E)_1(s)$$
$$\lambda X.\ (C\ Y\ E[s \mapsto X])_2(0)$$
$$\lambda Y.\ (C\ Y\ E)_2(0)$$

Ceci nous donne une spécification dans Coq qui a la forme suivante:

```
Inductive TRAD:
   Stmt ->
   (C:Circ)
   ((Y:ScottBool) (E:ScottAdic) (si:nat)
    (continue [X:ScottBool]((Fst (C Y (new E si X))) si))) ->
   ((E:ScottAdic) (si:nat)(continue [Y:ScottBool]((Fst (C Y E)) si))) ->
   ((Y:ScottBool) (E:ScottAdic) (si:nat)
    (continue [X:ScottBool]((Snd (C Y (new E si X))) O))) ->
   ((E:ScottAdic)(continue [Y:ScottBool]((Snd (C Y E)) O))) ->Prop :=
  :
  :
  :
```

4 Preuve de correction de la traduction

Nous avons développé la preuve de correction en suivant la deuxième approche vue dans le chapitre précédant.

Soit P un programme Esterel de corps p. Comme expliqué dans [2], pour calculer la réaction $P \xrightarrow[I]{O} P'$ d'un programme Esterel on procède en deux étapes.

1. On détermine O grace au prédicat MC.

2. On calcule le programme dérivé P' grace à la relation de transition $p \overset{O,k}{\underset{I \cup O}{\hookrightarrow}} p'$

Le prédicat MC suffit pour calculer l'événement de sortie d'une réaction et comme nous ne nous intéressons pas ici aux programmes dérivés, nous ne nous servirons pas de la relation de transition. De plus, nous nous plaçons dans le cas où les signaux de sortie (O) ne sont pas lus. Nous n'aurons donc pas d'itération suplémentaire au cours de la première étape, pour déterminer le statut d'éventuels "inputoutput".

Nous allons énoncer la preuve de correction de la traduction donnée en fig.6 en sémantique naturelle pour des raisons de lisibilité. Les deux lemmes suivants nous serons utiles dans cette preuve.

$$\text{Lem1} \ : \ \frac{p \ \rightarrow \ C}{(C \ \mathbf{0} \ E) \ = \ < \ \mathsf{Zero} \ , \ \mathsf{Zero} \ >}$$

$$\text{Lem2} \ : \ \frac{p \ \rightarrow \ C}{(C \ \perp \ E)_1(n) \neq \mathbf{1} \ \wedge \ (C \ \perp \ E)_2(n) \neq \mathbf{1}}$$

Ces deux lemmes se montre par induction sur la structure de la preuve de la proposition: $p \ \rightarrow \ C$.

Nous allons maintenant donner les grandes lignes de la preuve de correction de la traduction. Il s'agit de montrer:

$$\text{correct\_trad} \ : \ \frac{MC^{\alpha}(p, E) \ = \ < E', K > \qquad p \ \rightarrow \ C}{(C \ \alpha \ E) \ = \ < E', K >}$$

Ce théorème se montre par récurrence sur la structure de la preuve de la proposition: $MC^{\alpha}(p, E) =< E', K >$. Il faut étudier chacun des cas correspondant aux 15 règles d'introduction du prédicat MC.

- Le cas où $\alpha = \mathbf{0}$ se résoud trivialement en appliquant le lemme Lem1 énoncé plus haut. Il nous reste donc à étudier les cas pour lesquels $\alpha \neq \mathbf{0}$, que nous avons regroupés par instruction dans ce qui suit.

- $k \ , \ s!$

  Les cas traitant le code de terminaison et l'émission s'expriment par des axiomes des prédicats MC et $TRAD$, qui conduisent à deux égalités triviales.

- $s?p, q$

  Les deux premier cas du test de présence correspondant aux statut $\mathbf{1}$ et $\mathbf{0}$ se simplifient en utilisant le lemme Lem1 (pour q et p resp.), puis se résolvent en appliquant l'hypothèse de récurrence sur p et q suivant le cas.

  Le dernier cas correspondant au statut $\perp$ est également trivial car pour $\alpha \neq \mathbf{0}$ on a forcèment $\alpha \wedge \perp = \perp$, ce qui nous permet d'appliquer les deux hypothèses de récurrence sur p et q.

- $p; q$

  Trois cas interviennent ici.

  1. l'instruction p doit terminer et passe un contrôle $K_p(0)$ égal à $\mathbf{1}$ à l'instruction q. Soient P et Q les circuits qui implantent p et q. En appliquant l'hypothèse de récurrence sur p on déduit que $(P\ \alpha\ E)_2(0) = K_p(0) = \mathbf{1}$. Pour pouvoir appliquer les hypothèses de récurrence sur p et q il faut utiliser le fait que $\alpha = \mathbf{1}$. D'après le Lem2 on a $\forall n.(C\ \ \perp\ \ E)_2(n) \neq \mathbf{1}$, α ne peut donc pas être $\perp$ car $(P\ \alpha\ E)_2(0) = \mathbf{1}$. On a donc bien $\alpha = \mathbf{1}$, ce qui nous permet de finir la preuve en utilisant l'hypothèse de récurrence.
  2. l'instruction p ne peut pas terminer et passe un contrôle $K_p(0)$ égal à $\mathbf{0}$ à l'instruction q. On déduit comme dans le cas précédant que $(P\ \alpha\ E)_2(0) = K_p(0) = \mathbf{0}$. Les équations se simplifient alors facilement grace au lemme Lem1 et on résoud ce cas en appliquant l' hypothèse de récurrence sur p.
  3. l'instruction p passe un contrôle $K_p(0)$ égal à $\perp$ à l'instruction q. On déduit comme précédemment que $(P\ \alpha\ E)_2(0) = K_p(0) = \perp$. Il suffit alors d'appliquer les hypothèses de récurrence sur p et q.

- $p|q$

  Ce cas se résoud trivialement en appliquant les hypothèses de récurrence.

- $p\backslash s$

  Il existe trois cas suivant le statut de s retourné par l'appel récursif du prédicat MC. Dans chacun des cas, on utilise tout d'abord l'hypothèse de récurrence relative à l'appel récursif de MC (avec un statut $\perp$ pour s), afin de déduire que le statut de s retourné est le même que le point fixe calculé par le circuit. On applique alors l'hypothèse de récurrence qui concerne l'appel récursif de MC avec ce nouveau statut.

- $\{p\}$, $\uparrow p$

  Les cas correspondant à ces deux instructions sont résolus trivialement par application de l'hypothèse de récurrence.

5 Travaux futurs

Les extensions de la preuve de correction pourraient se réaliser en deux étapes. La première étape concernerait l'extension aux circuits avec registres en incluant les instructions d'attente ("pause") et de suspension ("suspend"). La deuxième étape s'attaquerait quant à elle aux problèmes de schizophrénie posés par l'instruction de boucle ("loop").

Dans la première étape, les types coinductifs disponibles dans Coq pourraient s'avérer trés utiles en permettant la manipulation de chaînes infinies, nécessaires à la description du comportement d'un programme réactif et du circuit correspondant.

```
Section Streams. (* The set of streams : definition *)
```

```
Variable A : Set.

CoInductive Set Stream   := Cons : A->Stream->Stream.

Definition head  :=
  [x:Stream]<A>Case x of [a:A][s:Stream]a end.
Definition tail  :=
  [x:Stream]<Stream>Case x of [a:A][s:Stream]s end.

(* Extensional Equality between two streams  *)
CoInductive EqSt  : Stream->Stream->Prop :=
            eqst : (s1,s2:Stream)
                   ((head s1)=(head s2))->
                   (EqSt (tail s1) (tail s2))
                    ->(EqSt s1 s2).


   :
   :
End Streams.
```

L'ensemble des séquences de réactions d'un programme pourraient ainsi être spécifiées par quelque chose comme:

```
(* événements *)
Definition Event :=  ScottAdic.
(* chaines infinies d'événements d'entrée et de sortie *)
Definition StrI := (Stream Event).
Definition StrO := (Stream Event).


CoInductive StmtBehavior: StrI -> Stmt -> StrO -> Prop :=
  Intro_StmtBehavior:
     (Sin:StrI)(p,p':Stmt)(Out:Event)(SOut:StrO)(k:nat)
        (TRANSITION (head SI) p p' Out k) ->
            (StmtBehavior (tail SI) p' Out) ->
                  (StmtBehavior SI p (cons Out SOut)).
```

Le comportement d'un circuit pourraient alors se formaliser par une fonction d'une infinité d'événements d'entrée vers une infinité d'événements de sortie.

Pour cela il faudrait tout d'abord étendre la définition des circuits aux fonctions qui calculent l'événement de sortie du circuit et son nouvel état de registre pour un événement d'entrée et un état des registres donnés.

On pourrait imaginer dans Coq quelque chose comme:

```
(* fils *)
```

```
Definition Wire := ScottBool.
Definition WireSet := ScottAdic.
(* registres *)
Definition Reg := ScottAdic.
(* circuits *)
Definition Circ: Set := Wire -> ...  -> Wire -> StrI -> Reg -> StrO*Wire*WireSet*Reg.

CoFixpoint CircBehavior: Circ ->  StrI -> Reg -> StrO :=
   [SI:StrI][R:Reg] <StrO>let E::SE = SI in
                    <StrO>let (C GO ... In) = (E',SEL,K,R') in
                      E'::(CircBehavior Circ SE  R')
```

Le théorème de correction s'exprimerait alors en utilisant l'égalité paresseuse (`EqSt`) sur les chaines infinies d'événement de sortie.

Références

[1] H. Barendregt:Lambda calculi with types. The Handbook of Logic in Computer Science, Vol II, ed. by S. Abramsky, Dov M. Gabbay and T.S.E. Maibaum, Oxford (1992)

[2] G. Berry: The Constructive Semantics of Pure Esterel. Draft Version. http://cma.cma.fr/Personnel/Berry.html.

[3] T. Coquand : Une Théorie des Constructions, Université Paris VII, Thèse de troisième cycle, 1985.

[4] T. Coquand , G. Huet: Constructions: A higher order proof system for mechanizing mathematics. EUROCAL'85, 1985.

[5] C. Cornes, J. Courant, J.F. Filliâtre, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, B. Werner: The Coq Proof Assistant Reference Manual, Version 5.10. Projet Coq, Inria-Rocquencourt and CNRS-ENS Lyon, France.

[6] C. Paulin-Mohring: Inductive Definitions in the System Coq. Rules and Properties. Pr. of the Int. Conf. on approche Typed Lambda Calculi and Applications, LNCS 664 (1993) 328–345

[7] B. Werner: Une Théorie des Constructions Inductives. Thèse Université Paris VII, May 1994.

[8] Codifying guarded definitions with recursive schemes. In BRA Workshop on Types for Proofs and Programs, June 1994. To appear in the LNCA series.

[9] E. Gimenez: The implementation of coinductive types in Coq: an experiment with the Alternating Bit Protocol. In BRA Workshop Types for Proofs and Programs, Turin, June 1995

Annexe A

Code coq pour la première approche

```
Section ProdScottBool.

Inductive prodT [A,B:Type] : Type := pairT : A -> B -> (prodT A B).

   Definition fstT := [A,B:Type][H:(prodT A B)]<A> Case H of [x:A][y:B]x end.
   Definition sndT := [A,B:Type][H:(prodT A B)]<B> Case H of [x:A][y:B]y end.


Fixpoint prodn[A:Type; n:nat]: Type :=
   (<Type>Case n of unitT [p:nat](prodT A  (prodn A p)) end).


Fixpoint projk[A:Type; a:A; k:nat; n:nat]: (prodn A (plus n k))->A :=
   (<[n:nat] (prodn A (plus n k))->A>Case n of
     (<[k:nat] (prodn A (plus O k))->A>Case k of
       [V:(prodn A (plus O O))]a
       [q:nat] [V:(prodn A (plus O (S q)))](fstT A (prodn A q) V) end)
     [p:nat] [V:(prodn A (plus (S p) k))]
     (projk A a k p (sndT A (prodn A (plus p k)) V)) end).


Definition ScottBooln: nat->Type := (prodn ScottBool).

Definition ScottBoolpj: (k:nat)(n:nat)(prodn ScottBool (plus n k))->ScottBool := (projk ScottBool bot).

End ProdScottBool.



Section Circuits.

Definition one: nat := (S O).
Definition two: nat := (S (S O)).

Inductive gate: nat -> nat ->Set :=
   g_V0: (m:nat)(gate O m)
  | g_V1: (m:nat)(gate O m)
  | g_Vb: (m:nat)(gate O m)
  | g_or: (n:nat)(gate (plus n n) n)
  | g_and: (n:nat)(gate (plus n n) n)
```

```
  | g_not: (n:nat)(gate n n)
  | g_buff: (n:nat)(gate n n)
  | g_dupl: (n:nat)(gate n (plus n n))
  | g_reduc: (n:nat)(gate n O)
  | g_comp: (n, m, p:nat) (gate n m) -> (gate m p) ->(gate n p)
  | g_tens:
      (n, m, p, q:nat) (gate n m) -> (gate p q) ->(gate (plus n p) (plus m q))
  | g_refl: (i,j,n,m:nat) (gate n m) ->(gate n m)
  | coer_nv: (k:nat) (n:nat)(gate (S n) n)
  | coer_add: (n:nat)(gate n (S n))
  | coer_toend: (k:nat) (n:nat)(gate (plus n (S k)) (plus (S n) k))
  | coer_toendp: (k:nat) (n:nat)(gate (plus n k) (S (plus n k))).

Fixpoint gate_sem[n:nat; m:nat; G:(gate n m)]: (ScottBooln n) ->(ScottBooln m) :=
   (<[n, m:nat] (ScottBooln n) ->(ScottBooln m)>Case G of
     [m:nat] [x:(ScottBooln O)](ConstScottBool m ze)
     [m:nat] [x:(ScottBooln O)](ConstScottBool m on)
     [m:nat] [x:(ScottBooln O)](ConstScottBool m bot)
     [n:nat] [x:(ScottBooln (plus n n))]
     (or_ScottBooln n (decomp_lScottBooln n n x) (decomp_rScottBooln n n x))
     [n:nat] [x:(ScottBooln (plus n n))]
     (and_ScottBooln n (decomp_lScottBooln n n x) (decomp_rScottBooln n n x))
     [n:nat] [x:(ScottBooln n)](not_ScottBooln n x)
     [n:nat] [x:(ScottBooln n)]x
     [n:nat] [x:(ScottBooln n)](append_ScottBooln n n x x)
     [n:nat] [x:(ScottBooln n)] final
     [n, m, p:nat] [G1:(gate n m)] [G2:(gate m p)] [x:(ScottBooln n)]
     (gate_sem m p G2 (gate_sem n m G1 x))
     [n, m, p, q:nat] [G1:(gate n m)] [G2:(gate p q)] [x:(ScottBooln (plus n p))]
     (append_ScottBooln
        m q (gate_sem n m G1 (decomp_lScottBooln n p x))
        (gate_sem p q G2 (decomp_rScottBooln n p x)))
     [n, m:nat] [G:(gate (S n) (S m))] [x:(ScottBooln (S n))]
     (ConstScottBool (S m) ze final)
     [k, n:nat] [x:(ScottBooln (S n))]
     (nv_ScottBooln k (fstT ScottBool (ScottBooln n) x) n (sndT ScottBool (ScottBooln n) x))
     [n:nat] [x:(ScottBooln n)](pairT ScottBool (ScottBooln n) bot x)
     [k, n:nat] [x:(ScottBooln (plus n (S k)))](toend k n x)
     [k, n:nat] [x:(ScottBooln (plus n k))](toendplus k n x) end).
End Circuits.


Section Trad.

Definition two: nat := (S (S O)).
```

```
Inductive trad: (N:nat) (M:nat) Stmt -> (gate (S M) (S (plus N M))) ->Prop :=
   trad_0:
     (N, M:nat)
     (trad
        N M (term 0)
        (g_comp
           (S M) one (S (plus N M))
           (g_tens one one M 0 (g_buff one) (g_reduc M))
           (g_tens one one 0 (plus N M) (g_buff one) (g_V0 (plus N M)))))
  | trad_sup2:
      (N, M:nat) (k:nat) ~ k =two ->
      (trad
         N M (term k)
         (g_comp
            (S M) one (S (plus N M))
            (g_tens one one M 0 (g_buff one) (g_reduc M))
            (g_comp
               one (S (S (plus N M))) (S (plus N M))
               (g_tens
                  one one 0 (S (plus N M)) (g_buff one) (g_V0 (S (plus N M))))
               (coer_nv k (S (plus N M))))))
  | trad_emit:
      (N, M:nat) (s:signal)
      (trad
         N M (emit s)
         (g_comp
            (S M) two (S (plus N M))
            (g_tens one two M 0 (g_dupl one) (g_reduc M))
            (g_comp
               two (S (S (plus N M))) (S (plus N M))
               (g_tens two two 0 (plus N M) (g_buff two) (g_V0 (plus N M)))
               (coer_nv (idx s) (S (plus N M))))))
  | trad_pre:
      (N:nat)
      (s:signal)
      (i, k:nat)
      (p, q:Stmt)
      (P, Q:(gate (S (plus i k)) (S (plus N (plus i k)))))
      (trad N (plus i k) p P) -> (trad N (plus i k) q Q) -> i =(idx s) ->
      (trad
         N (plus i k) (present s p q)
         (g_comp
            (S (plus i k)) (plus (S (plus i k)) (S (plus i k)))
            (S (plus N (plus i k))) (g_dupl (S (plus i k)))
            (g_comp
               (plus (S (plus i k)) (S (plus i k)))
```

```
              (plus (S (S (plus i k))) (S (S (plus i k))))
              (S (plus N (plus i k)))
              (g_tens
                 (S (plus i k)) (S (S (plus i k))) (S (plus i k))
                 (S (S (plus i k)))
                 (g_tens
                    one one (plus i k) (S (plus i k)) (g_buff one)
                    (coer_toendp k i))
                 (g_tens
                    one one (plus i k) (S (plus i k)) (g_buff one)
                    (coer_toendp k i)))
              (g_comp
                 (plus (S (S (plus i k))) (S (S (plus i k))))
                 (plus (S (plus i k)) (S (plus i k))) (S (plus N (plus i k)))
                 (g_tens
                    (S (S (plus i k))) (S (plus i k)) (S (S (plus i k)))
                    (S (plus i k))
                    (g_tens
                       two one (plus i k) (plus i k) (g_and one)
                       (g_buff (plus i k)))
                    (g_tens
                       two one (plus i k) (plus i k)
                       (g_comp
                          two two one
                          (g_tens one one one one (g_not one) (g_buff one))
                          (g_and one)) (g_buff (plus i k))))
                 (g_comp
                    (plus (S (plus i k)) (S (plus i k)))
                    (plus (S (plus N (plus i k))) (S (plus N (plus i k))))
                    (S (plus N (plus i k)))
                    (g_tens
                       (S (plus i k)) (S (plus N (plus i k))) (S (plus i k))
                       (S (plus N (plus i k))) P Q)
                    (g_or (S (plus N (plus i k)))))))))
| trad_seq:
    (N, M:nat)
    (p, q:Stmt)
    (P, Q:(gate (S M) (S (plus N M)))) (trad N M p P) -> (trad N M q Q) ->
    (trad
       N M (seq p q)
       (g_comp
          (S M) (plus (S M) (S M)) (S (plus N M)) (g_dupl (S M))
          (g_comp
             (plus (S M) (S M)) (plus (S (plus N M)) (S M)) (S (plus N M))
             (g_tens (S M) (S (plus N M)) (S M) (S M) P (g_buff (S M)))
             (g_comp
```

```
                (plus (S (plus N M)) (S M)) (plus (plus N M) (S M))
                (S (plus N M)) (coer_nv (S M) (plus (plus N M) (S M)))
                (g_comp
                   (plus (plus N M) (S M)) (plus (plus N M) (S (plus N M)))
                   (S (plus N M))
                   (g_tens
                      (plus N M) (plus N M) (S M) (S (plus N M))
                      (g_buff (plus N M)) Q)
                   (g_comp
                      (plus (plus N M) (S (plus N M)))
                      (plus (S (plus N M)) (plus N M)) (S (plus N M))
                      (coer_toend (plus N M) (plus N M))
                      (g_tens
                         one one (plus (plus N M) (plus N M)) (plus N M)
                         (g_buff one) (g_or (plus N M)))))))))
| trad_abs:
    (N, M:nat)
    (s:signal) (p:Stmt) (P:(gate (S M) (S (plus N M)))) (trad N M p P) ->
    (trad N M (abs s p) (g_refl (idx s) (idx s) N M P))
| trad_trap:
   (N, M:nat)
   (p:Stmt)
   (P:(gate (S M) (S (plus (plus N two) M)))) (trad (plus N two) M p P) ->
   (trad
      (plus N one) M (trap p)
      (g_comp
         (S M) (S (plus (plus N two) M)) (plus (S (plus N one)) M) P
         (g_comp
            (plus (S (plus N two)) M) (plus (S (plus (S N) one)) M)
            (plus (S (plus N one)) M)
            (g_tens
               (plus (S N) two) (S (plus (S N) one)) M M
               (coer_toend one (S N)) (g_buff M))
            (g_tens
               (S (S (plus N one))) (S (plus N one)) M M
               (g_tens
                  two one (plus N one) (plus N one) (g_or one)
                  (g_buff (plus N one))) (g_buff M))))).




End Trad.
```

Annexe B

Code Coq pour la seconde approche

Fichier “scott_def.v”

```
(*************************************************************************

                        Scott Booleans definition

*************************************************************************)

Inductive ScottBool: Set :=
   zero: ScottBool
  | one: ScottBool
  | bottom: ScottBool.

Definition Sor: ScottBool -> ScottBool ->ScottBool :=
   [a, b:ScottBool]
   (<ScottBool>Case a of b one (<ScottBool>Case b of bottom one bottom end) end).

Definition Sand: ScottBool -> ScottBool ->ScottBool :=
   [a, b:ScottBool]
   (<ScottBool>Case a of zero b (<ScottBool>Case b of zero bottom bottom end)
     end).

Definition Snot: ScottBool ->ScottBool :=
   [a:ScottBool](<ScottBool>Case a of one zero bottom end).

(*************************************************************************

                        Scott Adics definition

*************************************************************************)

Definition ScottAdic: Set := nat ->ScottBool.

Definition Or: ScottAdic -> ScottAdic ->ScottAdic :=
   [f, g:ScottAdic] [n:nat](Sor (f n) (g n)).

Definition And: ScottAdic -> ScottAdic ->ScottAdic :=
   [f, g:ScottAdic] [n:nat](Sand (f n) (g n)).
```

```
Definition Not: ScottAdic ->ScottAdic := [f:ScottAdic] [n:nat](Snot (f n)).

Definition Fst := (fst ScottAdic ScottAdic).

Definition Snd := (snd ScottAdic ScottAdic).

Definition Zero := [n:nat]zero.
(**************************************************************************

                          Scott CPO

**************************************************************************
** {f:ScottBool -> ScottBool} continue <-> monotone + finitaire <-> continue ***)

Inductive ScottRel: ScottBool -> ScottBool ->Prop :=
   rel_zero: (ScottRel zero zero)
  | rel_one: (ScottRel one one)
  | rel_bottom: (a:ScottBool)(ScottRel bottom a).

Definition monotone: (ScottBool ->ScottBool) ->Prop :=
   [f:ScottBool ->ScottBool] (a, b:ScottBool) (ScottRel a b) ->
   (ScottRel (f a) (f b)).

Definition continue: (ScottBool ->ScottBool) ->Prop :=
   [f:ScottBool ->ScottBool] ~ (f bottom) =bottom ->
   (f zero) =(f bottom) /\ (f one) =(f bottom).

Inductive lfp[f:ScottBool ->ScottBool; x:ScottBool]: Prop :=
   lfp_intro: (f x) =x -> ((y:ScottBool) (f y) =y ->(ScottRel x y)) ->(lfp f x).


(**************************************************************************

                    Auxiliary functions definitions

**************************************************************************)

Lemma Lemnat: (n, m:nat){n =m}+{~ n =m}.
Induction n; Induction m.
Left; Trivial.
Intros n0 H'; Right; Discriminate.
Right; Discriminate.
Intros n1 H'; Elim (H n1).
Intro H'0; Left; Rewrite H'0; Trivial.
Intro H'0; Right; Red; Intro H'1; Apply H'0; Injection H'1; Intros; Assumption.
```

```
Qed.

Definition New: ScottAdic -> nat -> ScottBool ->ScottAdic :=
   [f:ScottAdic] [i:nat] [x:ScottBool] [n:nat]
   (<ScottBool>Case (Lemnat n i) of [H:n =i]x [H:~ n =i](f n) end).
Axiom Lemle:(n, m:nat){(le n m)}+{(le m n)}.

Definition max: nat -> nat ->nat :=
   [n, m:nat](<nat>Case (Lemle n m) of [H:(le n m)]m [H:(le m n)]n end).

Fixpoint or_until[f:ScottAdic; n:nat]: ScottBool :=
   (<ScottBool>Case n of (f O) [m:nat](Sor (f (S m)) (or_until f m)) end).

Definition Max: ScottAdic -> ScottAdic ->ScottAdic :=
   [f, g:ScottAdic] [n:nat]
   (Sand (or_until f n) (Sand (Sor (f n) (g n)) (or_until g n))).

Definition down: nat ->nat :=
   [n:nat]
   (<nat>Case n of
     O [p:nat](<nat>Case p of (S O) [q:nat](<nat>Case q of O [r:nat]r end) end)
     end).

Definition Down: ScottAdic ->ScottAdic :=
   [f:ScottAdic] [n:nat]
   (<ScottBool>Case n of
     (Sor (f O) (f (S (S O))))
     [m:nat](<ScottBool>Case m of (f (S O)) [p:nat](f (S (S (S p)))) end) end).

Definition up: nat ->nat :=
   [n:nat]
   (<nat>Case n of
     O
     [p:nat]
     (<nat>Case p of
       (S O) [q:nat](<nat>Case q of (S (S (S O))) [r:nat](S (S r)) end) end)
     end).

Definition Up: ScottAdic ->ScottAdic :=
   [f:ScottAdic] [n:nat]
   (<ScottBool>Case n of
     (f O)
     [m:nat]
     (<ScottBool>Case m of
       (f (S O)) [p:nat](<ScottBool>Case p of zero [q:nat](f (S (S q))) end) end) end).
```

Fichier “scott_lem.v”

```
Require Import scott_def.

(*************************************************************************

                    Lemmas on Scott Booleans

*************************************************************************)

Lemma LemScottBool: (a, b:ScottBool){a =b}+{~ a =b}.
Intros a b; Case a; Case b; (Right; Discriminate) Orelse (Left; Trivial).
Qed.

Lemma Sor_com: (a, b:ScottBool)(Sor a b) =(Sor b a).
Intros a b; Case a; Case b; Trivial.
Qed.

Lemma Sand_com: (a, b:ScottBool)(Sand a b) =(Sand b a).
Intros a b; Case a; Case b; Trivial.
Qed.

Lemma notand_imp_not1_or_not2:
  (a, b:ScottBool) ~ (a =one /\ b =one) ->~ a =one \/ ~ b =one.
Intros a b; Case a; Case b; Intro H';
 (Left; Discriminate)
  Orelse (Right; Discriminate)
         Orelse (Unfold 1 not in H'; LApply H';
                  [Intro H'0; Elim H'0 | Split; Trivial]).
Qed.

Lemma not1_or_not2_imp_notand:
  (a, b:ScottBool) ~ a =one \/ ~ b =one ->~ (a =one /\ b =one).
Intros a b H'; Elim H';
 (Intro H'0; Clear H'; Red; Intro H'; Elim H'; Intros H'1 H'2;
   Apply H'0 Orelse Elim H'0; Try Assumption).
Qed.

Lemma notSand_imp_not1_or_not2:
  (a, b:ScottBool) ~ (Sand a b) =one ->~ a =one \/ ~ b =one.
Intros a b; Case a; Case b;
 (Intro H';
   (Left; Discriminate)
```

```
    Orelse (Right; Discriminate)
            Orelse (Generalize (H' (refl_equal ScottBool one)); Intro FF;
                    Elim FF)).
Qed.

Lemma not1_or_not2_imp_notSand:
  (a, b:ScottBool) ~ a =one \/ ~ b =one ->~ (Sand a b) =one.
Intros a b; Case a; Case b;
 (Intros;
   Discriminate
    Orelse (Elim H; Intro H'; Generalize (H' (refl_equal ScottBool one));
            Intro FF; Elim FF)).
Qed.

(*************************************************************************

                    Lemmas on Scott Adics

*************************************************************************)

Lemma Or_com: (f, g:ScottAdic) (n:nat)((Or f g) n) =((Or g f) n).
Unfold Or; Intros f g n; Apply (Sor_com (f n) (g n)).
Qed.

Lemma And_com: (f, g:ScottAdic) (n:nat)((And f g) n) =((And g f) n).
Unfold And; Intros f g n; Apply (Sand_com (f n) (g n)).
Qed.

Lemma Or_2Zero: (n:nat)((Or Zero Zero) n) =(Zero n).
Unfold Or; Intro n; Simpl; Trivial.
Qed.

Lemma Or_Zero: (n:nat) (F:ScottAdic)((Or Zero F) n) =(F n).
Unfold Or; Intros n F; Simpl; Trivial.
Qed.

(*************************************************************************

                            More Lemmas

*************************************************************************)

Lemma New_Zero_zero_eq_Zero: (k:nat) (n:nat)((New Zero k zero) n) =(Zero n).
Intros k n; Unfold 1 New; Case (Lemnat n k); Simpl; Trivial.
Qed.
```

```
Lemma New_diff_com:
  (si, sj:nat) (a, b:ScottBool) (E:ScottAdic) ~ si =sj ->
  (n:nat)((New (New E si a) sj b) n) =((New (New E sj b) si a) n).
Unfold New.
Intros si sj a b E H' n; Case (Lemnat n sj); Case (Lemnat n si);
 (Intro H'0; Rewrite H'0; Intro H'1; Generalize (H' H'1); Intro FF; Elim FF)
  Orelse (Intros H'0 H'1; Trivial).
Qed.

Lemma New_eq_com:
  (si:nat) (a, b:ScottBool) (E:ScottAdic) (n:nat)
  ((New (New E si a) si b) n) =((New E si b) n).
Intros si a b E n; Unfold New ; Case (Lemnat n si); Case (Lemnat n si);
 Intros H' H'0; Trivial.
Qed.

Lemma Lem_or_until:
  (A, B:ScottAdic) ((n:nat)<ScottBool> (A n) =(B n)) ->
  (n:nat)<ScottBool> (or_until A n) =(or_until B n).
Intros A B H' n; Elim n; Simpl.
Specialize 1 H' with n := O; Intro H'0; Try Exact H'0.
Intros n0 H'0; Rewrite H'0; Rewrite (H' (S n0)); Trivial.
Qed.

Lemma Lem_Down:
  (A, B:ScottAdic) ((n:nat)<ScottBool> (A n) =(B n)) ->
  (n:nat)<ScottBool> (Down A n) =(Down B n).
Intros A B H' n; Elim n; Simpl.
Rewrite (H' O); Rewrite (H' (S (S O))); Trivial.
Intros n0 H'0; Case n0.
Specialize 1 H' with n := (S O); Intro H'1; Try Exact H'1.
Intro n1; Try Exact n1.
Specialize 1 H' with n := (S (S (S n1))); Intro H'1; Try Exact H'1.
Qed.

Lemma Lem_Up:
  (A, B:ScottAdic) ((n:nat)<ScottBool> (A n) =(B n)) ->
  (n:nat)<ScottBool> (Up A n) =(Up B n).
Intros A B H' n; Elim n; Simpl.
Specialize 1 H' with n := O; Intro H'0; Try Exact H'0.
Intros n0 H'0; Case n0.
Specialize 1 H' with n := (S O); Intro H'1; Try Exact H'1.
Intro n1; Case n1; Trivial.
Qed.

Lemma Lem_Max:
```

```
  (A, B, A', B':ScottAdic)
  ((n:nat)<ScottBool> (A n) =(A' n)) -> ((n:nat)<ScottBool> (B n) =(B' n)) ->
  (n:nat)<ScottBool> (Max A B n) =(Max A' B' n).
Intros A B A' B' H' H'0 n; Case n.
Unfold Max.
Specialize 1 H' with n := O; Intro H'1; Rewrite H'1.
Specialize 1 H'0 with n := O; Intro H'2; Rewrite H'2.
LApply (Lem_or_until A A');
 [Intro H'5; Specialize 1 H'5 with n := O; Intro H'6; Rewrite H'6 | Trivial].
LApply (Lem_or_until B B');
 [Intro H'7; Specialize 1 H'7 with n := O; Intro H'8; Rewrite H'8 | Trivial].
Trivial.
Unfold Max.
Intro n0; Try Assumption.
Specialize 1 H' with n := (S n0); Intro H'1; Rewrite H'1.
Specialize 1 H'0 with n := (S n0); Intro H'2; Rewrite H'2.
LApply (Lem_or_until A A');
 [Intro H'5; Specialize 1 H'5 with n := (S n0); Intro H'6; Rewrite H'6 | Trivial].
LApply (Lem_or_until B B');
 [Intro H'7; Specialize 1 H'7 with n := (S n0); Intro H'8; Rewrite H'8 | Trivial].
Trivial.
Qed.
```

Fichier “trad_def.v”

```
Require Import scott_def.

(*************************************************************************

          Definition de la syntaxe et de la semantique d'Esterel

*************************************************************************
****** syntaxe ******)

Inductive Stmt: Set :=
   completion: nat ->Stmt
  | emit: nat ->Stmt
  | present: nat -> Stmt -> Stmt ->Stmt
  | seq: Stmt -> Stmt ->Stmt
  | parallel: Stmt -> Stmt ->Stmt
  | local: nat -> Stmt ->Stmt
  | trap: Stmt ->Stmt
  | shift: Stmt ->Stmt.
(****    Definition des predicats MUST et CAN par une fonction MC unique **
```

```
*************************************************************************)

Inductive MC: ScottBool -> Stmt -> ScottAdic -> ScottAdic -> ScottAdic ->Prop :=
   MC_alpha0: (E:ScottAdic) (p:Stmt)(MC zero p E Zero Zero)
  | MC_compl:
      (alpha:ScottBool) (k:nat) (E:ScottAdic) ~ alpha =zero ->
      (MC alpha (completion k) E Zero (New Zero k alpha))
  | MC_emit:
      (alpha:ScottBool) (si:nat) (E:ScottAdic) ~ alpha =zero ->
      (MC alpha (emit si) E (New Zero si alpha) (New Zero O alpha))
  | MC_present1:
      (alpha:ScottBool)
      (si:nat)
      (p, q:Stmt)
      (E, E', K:ScottAdic)
      ~ alpha =zero -> (E si) =one -> (MC alpha p E E' K) ->
      (MC alpha (present si p q) E E' K)
  | MC_present2:
      (alpha:ScottBool)
      (si:nat)
      (p, q:Stmt)
      (E, E', K:ScottAdic)
      ~ alpha =zero -> (E si) =zero -> (MC alpha q E E' K) ->
      (MC alpha (present si p q) E E' K)
  | MC_present3:
      (alpha:ScottBool)
      (si:nat)
      (p, q:Stmt)
      (E, Ep, Eq, Kp, Kq:ScottAdic)
      ~ alpha =zero ->
      (E si) =bottom -> (MC bottom p E Ep Kp) -> (MC bottom q E Eq Kq) ->
      (MC alpha (present si p q) E (Or Ep Eq) (Or Kp Kq))
  | MC_seq1:
      (alpha:ScottBool)
      (p, q:Stmt)
      (E, Ep, Eq, Kp, Kq:ScottAdic)
      ~ alpha =zero ->
      (MC alpha p E Ep Kp) -> (Kp O) =one -> (MC alpha q E Eq Kq) ->
      (MC alpha (seq p q) E (Or Ep Eq) (Or (New Kp O zero) Kq))
  | MC_seq2:
      (alpha:ScottBool)
      (p, q:Stmt)
      (E, Ep, Eq, Kp, Kq:ScottAdic)
      ~ alpha =zero -> (MC alpha p E Ep Kp) -> (Kp O) =zero ->
      (MC alpha (seq p q) E Ep Kp)
  | MC_seq3:
```

```
      (alpha:ScottBool)
      (p, q:Stmt)
      (E, Ep, Eq, Kp, Kq:ScottAdic)
      ~ alpha =zero ->
      (MC alpha p E Ep Kp) -> (Kp 0) =bottom -> (MC bottom q E Eq Kq) ->
      (MC alpha (seq p q) E (Or Ep Eq) (Or (New Kp 0 zero) Kq))
  | MC_parallel:
      (alpha:ScottBool)
      (p, q:Stmt)
      (E, Ep, Eq, Kp, Kq:ScottAdic)
      ~ alpha =zero -> (MC alpha p E Ep Kp) -> (MC alpha q E Eq Kq) ->
      (MC alpha (parallel p q) E (Or Ep Eq) (Max Kp Kq))
  | MC_local1:
      (alpha:ScottBool)
      (si:nat)
      (p:Stmt)
      (E, Ep, Kp, Ep', Kp':ScottAdic)
      ~ alpha =zero ->
      (MC alpha p (New E si bottom) Ep Kp) ->
      (Ep si) =zero -> (MC alpha p (New E si zero) Ep' Kp') ->
      (MC alpha (local si p) E Ep' Kp')
  | MC_local2:
      (si:nat)
      (p:Stmt)
      (E, Ep, Kp, Ep', Kp':ScottAdic)
      (MC one p (New E si bottom) Ep Kp) ->
      (Ep si) =one -> (MC one p (New E si one) Ep' Kp') ->
      (MC one (local si p) E Ep' Kp')
  | MC_local3:
      (si:nat)
      (p:Stmt)
      (E, Ep, Kp:ScottAdic)
      (MC bottom p (New E si bottom) Ep Kp) -> (Ep si) =bottom ->
      (MC bottom (local si p) E Ep Kp)
  | MC_trap:
      (alpha:ScottBool)
      (p:Stmt) (E, Ep, Kp:ScottAdic) ~ alpha =zero -> (MC alpha p E Ep Kp) ->
      (MC alpha (trap p) E Ep (Down Kp))
  | MC_shift:
      (alpha:ScottBool)
      (p:Stmt) (E, Ep, Kp:ScottAdic) ~ alpha =zero -> (MC alpha p E Ep Kp) ->
      (MC alpha (shift p) E Ep (Up Kp)).

(****   Definition de la semantique constructive comportementale      ***
***  du noyau d'Esterel              ***
*************************************************************************)
```

```
Inductive TRANSITION: ScottAdic -> Stmt -> Stmt -> ScottAdic -> nat ->Prop :=
   trans_compl:
     (E:ScottAdic) (k:nat)(TRANSITION E (completion k) (completion 0) Zero k)
  | trans_emit:
     (si:nat) (E:ScottAdic)
     (TRANSITION E (emit si) (completion 0) (New Zero si one) 0)
  | trans_present1:
     (si:nat)
     (E:ScottAdic)
     (E':ScottAdic)
     (p, p', q:Stmt) (k:nat) (E si) =one -> (TRANSITION E p p' E' k) ->
     (TRANSITION E (present si p q) p' E' k)
  | trans_present2:
     (si:nat)
     (E:ScottAdic)
     (E':ScottAdic)
     (p, q, q':Stmt) (k:nat) (E si) =zero -> (TRANSITION E q q' E' k) ->
     (TRANSITION E (present si p q) q' E' k)
  | trans_seq1:
     (p, p', q, q':Stmt)
     (E:ScottAdic)
     (Ep, Eq:ScottAdic)
     (k:nat) (TRANSITION E p p' Ep 0) -> (TRANSITION E q q' Eq k) ->
     (TRANSITION E (seq p q) q' (Or Ep Eq) k)
  | trans_seq2:
     (p, p', q:Stmt)
     (E:ScottAdic) (E':ScottAdic) (k:nat) (TRANSITION E p p' E' k) -> ~ k =0 ->
     (TRANSITION E (seq p q) (seq p' q) E' k)
  | trans_parallel:
     (p, p', q, q':Stmt)
     (E:ScottAdic)
     (Ep, Eq:ScottAdic)
     (k, k':nat) (TRANSITION E p p' Ep k) -> (TRANSITION E q q' Eq k') ->
     (TRANSITION E (parallel p q) (parallel p' q') (Or Ep Eq) (max k k'))
  | trans_local1:
     (si:nat)
     (E, Ep, K:ScottAdic)
     (p, p':Stmt)
     (E':ScottAdic)
     (k:nat)
     (MC one p (New E si bottom) Ep K) ->
     (Ep si) =one -> (TRANSITION (New E si one) p p' E' k) ->
     (TRANSITION E (local si p) (local si p') (New E' si zero) k)
  | trans_local2:
     (si:nat)
```

```
      (E, Ep, K:ScottAdic)
      (p, p':Stmt)
      (E':ScottAdic)
      (k:nat)
      (MC one p (New E si bottom) Ep K) ->
      (Ep si) =zero -> (TRANSITION (New E si zero) p p' E' k) ->
      (TRANSITION E (local si p) (local si p') E' k)
  | trans_trap1:
      (p, p':Stmt)
      (E:ScottAdic)
      (E':ScottAdic) (k:nat) (TRANSITION E p p' E' k) -> k =0 \/ k =(S (S 0)) ->
      (TRANSITION E (trap p) (completion 0) E' 0)
  | trans_trap2:
      (p, p':Stmt)
      (E:ScottAdic)
      (E':ScottAdic) (k:nat) (TRANSITION E p p' E' k) -> (gt k (S (S 0))) ->
      (TRANSITION E (trap p) (trap p') E' (down k))
  | trans_shift:
      (p, p':Stmt)
      (E:ScottAdic) (E':ScottAdic) (k:nat) (TRANSITION E p p' E' k) ->
      (TRANSITION E (trap p) (shift p') E' (up k)).

(*************************************************************************

                Definition de la traduction en circuits

*************************************************************************
** Circ = (GO x E) -> (E' x K) ***)

Definition Circ: Set := ScottBool -> ScottAdic ->ScottAdic * ScottAdic.

Inductive TRAD: Stmt -> Circ ->Prop :=
   trad_compl:
     (k:nat)
     (TRAD
        (completion k)
        [GO:ScottBool] [E:ScottAdic]
        <ScottAdic, ScottAdic> (Zero, (New Zero k GO)))
  | trad_emit:
      (si:nat)
      (TRAD
         (emit si)
         [GO:ScottBool] [E:ScottAdic]
         <ScottAdic, ScottAdic> ((New Zero si GO), (New Zero 0 GO)))
  | trad_present:
      (si:nat) (p, q:Stmt) (P, Q:Circ) (TRAD p P) -> (TRAD q Q) ->
```

```
    (TRAD
       (present si p q)
       [GO:ScottBool] [E:ScottAdic]
       <ScottAdic, ScottAdic>
           ((Or
               (Fst (P (Sand GO (E si)) E))
               (Fst (Q (Sand GO (Snot (E si))) E))),
            (Or
               (Snd (P (Sand GO (E si)) E))
               (Snd (Q (Sand GO (Snot (E si))) E)))))
| trad_seq:
    (p, q:Stmt) (P, Q:Circ) (TRAD p P) -> (TRAD q Q) ->
    (TRAD
       (seq p q)
       [GO:ScottBool] [E:ScottAdic]
       <ScottAdic, ScottAdic>
           ((Or (Fst (P GO E)) (Fst (Q ((Snd (P GO E)) O) E))),
            [n:nat]
            (<ScottBool>Case n of
              ((Snd (Q ((Snd (P GO E)) O) E)) O)
              [m:nat]
              ((Or (Snd (P GO E)) (Snd (Q ((Snd (P GO E)) O) E))) (S m)) end)))
| trad_parallel:
    (p, q:Stmt) (P, Q:Circ) (TRAD p P) -> (TRAD q Q) ->
    (TRAD
       (parallel p q)
       [GO:ScottBool] [E:ScottAdic]
       <ScottAdic, ScottAdic>
           ((Or (Fst (P GO E)) (Fst (Q GO E))),
            (Max (Snd (P GO E)) (Snd (Q GO E)))))
| trad_local:
    (si:nat) (p:Stmt) (P:Circ) (TRAD p P) ->
    (TRAD
       (local si p)
       [GO:ScottBool] [E:ScottAdic]
       (P GO (New E si ((Fst (P GO (New E si bottom))) si))))
| trad_trap:
    (p:Stmt) (P:Circ) (TRAD p P) ->
    (TRAD
       (trap p)
       [GO:ScottBool] [E:ScottAdic]
       <ScottAdic, ScottAdic> ((Fst (P GO E)), (Down (Snd (P GO E)))))
| trad_shift:
    (p:Stmt) (P:Circ) (TRAD p P) ->
    (TRAD
       (shift p)
```

```
        [GO:ScottBool] [E:ScottAdic]
        <ScottAdic, ScottAdic> ((Fst (P GO E)), (Up (Snd (P GO E))))).
```

Fichier “theorems.v”

```
(*************************************************************************

                    Theorems : invariants of the circuits

*************************************************************************)



Theorem GO_eq_ze_imp_C_eq_ze:
  (p:Stmt) (C:Circ) (TRAD p C) ->
  (E:ScottAdic) (n:nat)((Fst (C zero E)) n) =zero /\ ((Snd (C zero E)) n) =zero.



Theorem GO_eq_bot_imp_C_neq_one:
  (p:Stmt) (C:Circ) (TRAD p C) ->
  (E:ScottAdic) (n:nat)
  ~ ((Fst (C bottom E)) n) =one /\ ~ ((Snd (C bottom E)) n) =one.



Theorem MC_Circ:
  (GO:ScottBool) (E, E', K:ScottAdic) (p:Stmt) (MC GO p E E' K) ->
  (C:Circ) (TRAD p C) ->
  (n:nat)((Fst (C GO E)) n) =(E' n) /\ ((Snd (C GO E)) n) =(K n).
```

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399